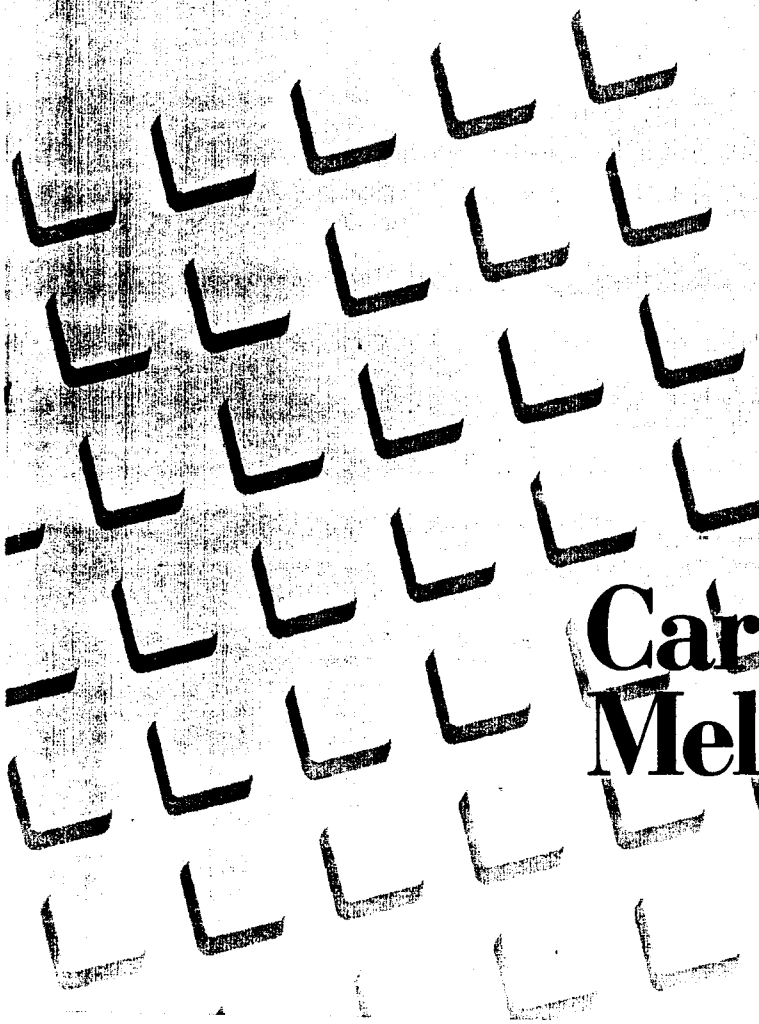
Problems in Rewriting applied to Categorical Concepts by the Example of a Computational Comonad

Wolfgang Gehrke

October 1994
CMU-CS-94-207



Carnegie Mellon

Problems in Rewriting applied to Categorical Concepts by the Example of a Computational Comonad

Wolfgang Gehrke

October 1994
CMU-CS-94-207

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present a canonical system for comonads which can be extended to the notion of a computational comonad [BG91] where the crucial point is to find an appropriate representation. These canonical systems are checked with the help of the Larch Prover [GG91] exploiting a method by G. Huet [Hue90a] to represent typing within an untyped rewriting system. The resulting decision procedures are implemented in the programming language Elf [Pfe89] since typing is directly supported by this language. Finally we outline an incomplete attempt to solve the problem which could be used as a benchmark for rewriting tools.

This research was sponsored in part by the National Science Foundation under Grant No. CCR-9303383, and in part by the Austrian Science Foundation (FWF), under ESPRIT BRP 6471 MEDLAR II.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of neither NSF nor the Austrian Science Foundation (FWF).

Keywords: algebraic approaches to semantics, denotational semantics, decision problems

1 Introduction

The starting point of this work was to provide methods for checking the commutativity of diagrams arising in category theory. Diagrams in this context are used as a visual description of equations between morphisms. To check the commutativity of a diagram amounts to check the equality of the morphisms involved. One way to support this task is to solve the uniform word problem for this category. Of course this is not always decidable.

A monoid is a very useful mathematical notion which is described equationally and which has a decidable uniform word problem. The equations can be characterized by diagrams as in Figure 1. A monad is the categorical generalization of the this concept [ML71]. This gives evidence that there can be a canonical system for monads, too.

Recently the concept of a monad became important also in computer science. In [Mog89] it was used to structure the semantics of programming languages which later was applied to structure purely functional programs as in [Wad93].

In [Geh94] we reduced the uniform word problem for monads to the uniform word problem for adjunctions exploiting theorems from category theory [BW85]. But when doing the same for a comonad the resulting system could not be extended to the notion of a computational comonad [BG91] which is used to study the intensional semantics of programming languages.

In this paper we describe a canonical system for computational comonads which is based on a different canonical system for comonads. It seems that the main difficulty in handling these equational theories consists of an appropriate reformulation of the problem with the help of an equivalent theory where Cartesian closed categories are a well known instance [Hue90a]. In that case the decision problem is transferred from the language of a CCC into the language of a typed lambda calculus.

Morphisms in a category come together with a type depending on two objects – source and target. These have to be taken into account when doing rewriting. In the verification of the canonical system we will deal with types in the frame of the Larch Prover [GG91] as suggested in [Hue90a]. Finally the resulting decision procedures were implemented in the programming language Elf [Pfe89] which directly supports dependent types and they were applied to examples from [BG91] dealing with computational comonads.

At the end we will briefly describe another attempt to solve the same problem with the help of rewriting over a congruence. Since for this congruence known methods do not apply and unification modulo a homomorphism is not enough we give a description formalizing powers of the endofunctor. This requires to handle at least addition of natural numbers and becomes very hard for rewriting.

The main contributions of this paper consist of:

- a representation of comonads which is suitable for rewriting and can be extended to the notion of a computational comonad providing a canonical system,
- the presentation of the method for encoding types within an untyped framework by using the example of a computational comonad,
- another demonstration of the usability of Elf as a tool for typed rewriting.

At the end we conclude and suggest future work.

2 Definitions

In this section the definition of a comonad [BW85] and a computational comonad [BG91] are given. Furthermore the notion of the Kleisli category is introduced. We assume throughout that the morphisms between two objects form a set.

Definition 1 (Comonad). Let C be a category. A *comonad* $CM = (T, \epsilon, \delta)$ on C is an endofunctor $T : C \to C$ with two natural transformations $\epsilon : T \to I_C$ and $\delta : T \to (T \circ T)$ where the following laws are satisfied:

$$\begin{array}{ll} \textbf{(Com1)} & T(\delta_A) \circ \delta_A = \delta_{T(A)} \circ \delta_A \\ \textbf{(Com2)} & T(\epsilon_A) \circ \delta_A = id_{T(A)} \\ \textbf{(Com3)} & \epsilon_{T(A)} \circ \delta_A = id_{T(A)} \end{array}$$

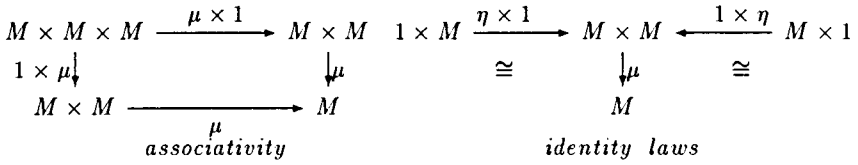


Fig. 1. laws of a monoid in form of diagrams

The laws for a comonad can be visualized by diagrams as in Figure 2 which is very similar to the diagrams of a monoid cf. Figure 1 (a real correspondence can be better seen for monads).

Note 2. Let C be a category. A comonad $CM = (T, \epsilon, \delta)$ can be characterized by the following equational specification:

$$id_B \circ f_{A \to B} = f \tag{1}$$
$$f_{A \to B} \circ id_A = f \tag{2}$$
$$(f_{A3 \to A4} \circ g_{A2 \to A3}) \circ h_{A1 \to A2} = f \circ (g \circ h) \tag{3}$$
$$T(id_A) = id_{T(A)} \tag{4}$$
$$T(f_{A2 \to A3} \circ g_{A1 \to A2}) = T(f) \circ T(g) \tag{5}$$
$$\epsilon_B \circ T(f_{A \to B}) = f \circ \epsilon_A \tag{6}$$
$$T(T(f_{A \to B})) \circ \delta_A = \delta_B \circ T(f) \tag{7}$$
$$T(\delta_A) \circ \delta_A = \delta_{T(A)} \circ \delta_A \tag{8}$$
$$T(\epsilon_A) \circ \delta_A = id_{T(A)} \tag{9}$$
$$\epsilon_{T(A)} \circ \delta_A = id_{T(A)} \tag{10}$$

Definition 3 (Kleisli category). Let $CM = (T, \epsilon, \delta)$ be a comonad on the category C. Then the corresponding *Kleisli category* K is defined by:

objects	same objects as C
morphisms	$Hom_K(A, B) = Hom_C(T(A), B)$
identity of object A	ϵ_A
composition of $f \in Hom_K(A2, A3)$ and $g \in Hom_K(A1, A2)$	$(f \circ_K g) := f \circ_C T(g) \circ_C \delta_{A1}$

Remark. It can easily be verified that this construction actually is a category cf. [BW85]. We present it since it will be useful for characterizing comonads in another way.

Definition 4 (Computational Comonad [BG91]). Let C be a category and $CM = (T, \epsilon, \delta)$ be a comonad on C. A *computational comonad* $CCM = (T, \epsilon, \delta, \gamma)$ is a comonad having additionally one further natural

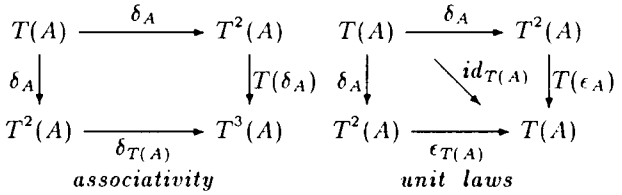


Fig. 2. comonad laws

transformation $\gamma : I_C \rightarrow T$ fulfilling the laws:

$$\begin{array}{ll} \textbf{(CCom1)} & \epsilon_A \circ \gamma_A = id_A \\ \textbf{(CCom2)} & \delta_A \circ \gamma_A = \gamma_{T(A)} \circ \gamma_A \end{array}$$

Note 5. Let C be a category. A computational comonad $CCM = (T, \epsilon, \delta, \gamma)$ can be characterized by an equational specification with the further equations:

$$T(f_{A \rightarrow B}) \circ \gamma_A = \gamma_B \circ f \tag{11}$$

$$\epsilon_A \circ \gamma_A = id_A \tag{12}$$

$$\delta_A \circ \gamma_A = \gamma_{T(A)} \circ \gamma_A \tag{13}$$

3 An Extensible Canonical System for a Comonad

In this section a rewriting system for comonads is presented and proved to be canonical. This particular representation of comonads is suitable to be extended to the notion of a computational comonad where a canonical system can be achieved, too.

Remark. The problem in using the original equational specification of a comonad comes from the fact that T is an endofunctor. This means that it can be iterated. But iteration requires the treatment of integer exponents which becomes difficult. Another problem is that ϵ interacts with T but δ interacts with $T \circ T$. This difference gives rise to complications, too.

Note 6. How can one take advantage from the Kleisli category which is formulated in terms of the given category? The law for multiplication in the Kleisli category suggests another auxiliary definition: $g^{\star}_{T(A) \rightarrow B} := T(g) \circ_C \delta_A$ (called the Kleisli star) such that $f \circ_K g = f \circ_C g^{\star}$. Formulating the categorical laws for the Kleisli category with this new function we get:

$$\begin{array}{rl} in \;\; K & in \;\; C \\ f = id_{(K)} \circ_K f = & \epsilon \circ_C f^{\star} \\ f = f \circ_K id_{(K)} = & f \circ_C \epsilon^{\star} \\ (f \circ_K g) \circ_K h & = (f \circ_C g^{\star}) \circ_C h^{\star} \\ \| & \| \\ f \circ_K (g \circ_K h) & = f \circ_C (g \circ_C h^{\star})^{\star} \end{array}$$

With setting $\upsilon_A := \epsilon_A$ (called the unit of the Kleisli category) these equations can be reformulated as:

$$\begin{array}{ll} \textbf{(CKl1)} & \upsilon_B \circ f^{\star}_{T(A) \rightarrow B} = f \\ \textbf{(CKl2)} & \upsilon^{\star}_A = id_{T(A)} \\ \textbf{(CKl3)} & g^{\star}_{T(A2) \rightarrow A3} \circ h^{\star}_{T(A1) \rightarrow A2} = (g \circ h^{\star})^{\star} \end{array}$$

Lemma 7. *A comonad can be completely characterized by $(id, \circ, \upsilon, {}^{\star})$ on the level of morphisms assuming it is known how T acts on objects.*

Proof cf. [Man76]. The other components can be expressed as:

$$\begin{aligned} \epsilon_A &:= \upsilon_A \\ \delta_A &:= id^{\star}_{T(A)} \\ T(f_{A \rightarrow B}) &:= (f \circ \upsilon_A)^{\star} \end{aligned}$$

□

Remark. This gives a more compact way of presenting a comonad since it was previously described in terms of $(id, \circ, T, \epsilon, \delta)$. Nevertheless the action of T on the object level has to be given since it plays a role in the typing information in the rules CKl1-3. This presentation can now be used to formulate a canonical system for comonads.

Proposition 8. *Assuming it is known how $T : C \to C$ acts on objects there is the following canonical system COM for a comonad:*

$$id_B \circ f_{A\to B} \longrightarrow f \tag{1}$$
$$f_{A\to B} \circ id_A \longrightarrow f \tag{2}$$
$$(f_{A3\to A4} \circ g_{A2\to A3}) \circ h_{A1\to A2} \longrightarrow f \circ (g \circ h) \tag{3}$$
$$v_A^\star \longrightarrow id_{T(A)} \tag{4}$$
$$v_B \circ f^\star_{T(A)\to B} \longrightarrow f \tag{5}$$
$$f^\star_{T(A2)\to A3} \circ g^\star_{T(A1)\to A2} \longrightarrow (f \circ g^\star)^\star \tag{6}$$
$$v_{A3} \circ (f^\star_{T(A2)\to A3} \circ g_{A1\to T(A2)}) \longrightarrow f \circ g \tag{7}$$
$$f^\star_{T(A3)\to A4} \circ (g^\star_{T(A2)\to A3} \circ h_{A1\to T(A2)}) \longrightarrow (f \circ g^\star)^\star \circ h \tag{8}$$

Proof. The correctness of this result was verified with the help of the Larch Prover. The check of the critical pairs can be seen in the appendix. It was done twice: firstly without taking the typing of morphisms into account and secondly including the typing. Here we only give the termination argument with the help of a polynomial interpretation $\mathcal{I}$ (cf. [Lan79]):

$$\begin{aligned}
\mathcal{I}_1(id) = \mathcal{I}_2(id) &= \mathcal{I}_3(id) = 1\\
\mathcal{I}_1(f \circ g) &= \mathcal{I}_1(f) + \mathcal{I}_1(g)\\
\mathcal{I}_2(f \circ g) &= 2 * \mathcal{I}_2(f) * \mathcal{I}_2(g)\\
\mathcal{I}_3(f \circ g) &= 2 * \mathcal{I}_3(f) + \mathcal{I}_3(g)\\
\mathcal{I}_1(v) = \mathcal{I}_2(v) &= \mathcal{I}_3(v) = 1\\
\mathcal{I}_1(f^\star) &= \mathcal{I}_1(f) + 2\\
\mathcal{I}_2(f^\star) &= \mathcal{I}_2(f) + 2\\
\mathcal{I}_3(f^\star) &= \mathcal{I}_3(f) + 2\\
\mathcal{I}(f) &:= (\mathcal{I}_1(f), \mathcal{I}_2(f), \mathcal{I}_3(f))
\end{aligned}$$

These triples of natural numbers are ordered lexicographically where the first component has the highest priority. Also this ordering is expressible in the Larch Prover and is given in the appendix. In the next section we will give more details about the typing. □

Remark. The termination argument in this case did not need any information from the typing. This is different to typed λ-calculus where mainly the types are used to prove termination. Starting from the canonical system COM we can try to extend the result to a computational comonad.

Note 9. How can the additional equations for a computational comonad be reformulated to fit into the new representation? Especially the references to T and δ have to be avoided. The key observation comes from the following equation:

$$f^\star_{T(A)\to B} \circ \gamma_A = T(f) \circ \delta_A \circ \gamma_A = T(f) \circ \gamma_{T(A)} \circ \gamma_A = \gamma_B \circ f \circ \gamma_A$$

Here both rules from the original presentation which involve T and δ have been applied leading to the single equation:

$$f^\star_{T(A)\to B} \circ \gamma_A = \gamma_B \circ f \circ \gamma_A$$

This equation together with $v_A \circ \gamma_A = id_A$ suffices to describe a computational comonad.

Lemma 10. *A computational comonad can be completely characterized by $(id, \circ, v, {}^\star, \gamma)$ on the level of morphisms assuming it is known how T acts on objects where γ is described by the equations:*

(Gam1) $v_A \circ \gamma_A = id_A$
(Gam2) $f^\star_{T(A)\to B} \circ \gamma_A = \gamma_B \circ f \circ \gamma_A$

Proof. It has to be shown that the previous three equations for γ can be derived:

$$T(f_{A\to B}) \circ \gamma_A = (f \circ \upsilon_A)^\star \circ \gamma_A = \gamma_B \circ (f \circ \upsilon_A) \circ \gamma_A = \gamma_B \circ f$$
$$\epsilon_A \circ \gamma_A = \upsilon_A \circ \gamma_A = id_{T(A)}$$
$$\delta_A \circ \gamma_A = id^\star_{T(A)} \circ \gamma_A = \gamma_{T(A)} \circ id_{T(A)} \circ \gamma_A = \gamma_{T(A)} \circ \gamma_A$$

□

Theorem 11. *Assuming it is know how $T : C \to C$ acts on objects there is the following canonical system CCOM for a computational comonad extending the canonical system COM*

$$\upsilon_A \circ \gamma_A \longrightarrow id_A \tag{9}$$
$$f^\star_{T(A)\to B} \circ \gamma_A \longrightarrow \gamma_B \circ (f \circ \gamma_A) \tag{10}$$
$$\upsilon_B \circ (\gamma_B \circ f_{A\to B}) \longrightarrow f \tag{11}$$
$$g^\star_{T(A2)\to A3} \circ (\gamma_{A2} \circ h_{A1\to A2}) \longrightarrow \gamma_{A3} \circ (g \circ (\gamma_{A2} \circ h)) \tag{12}$$

□

Proof. Again the check of the critical pairs was done in the Larch Prover which can be seen in the appendix. For the typing we refer to the next section. The previous polynomial interpretation was extended by:

$$\mathcal{I}_1(\gamma) = \mathcal{I}_2(\gamma) = \mathcal{I}_3(\gamma) = 1$$

□

Remark. At this point it should be stressed that the choice of the representation for comonads was not obvious to us. The introduction of the auxiliary * was necessary to succeed. This example could be used to test rewrite tools which allow the extension of the signature as in [KZ89] to which extent this may be automated.

4 Encoding Types in LP

Here the method due to G. Huet (cf. [Hue90a]) is demonstrated with the concrete example of a computational comonad.

Remark. In a category it is important to check the compatibility of morphisms in order to compose them. When we assume to start with compatible morphisms the application of the three untyped rules describing the categorical axioms coincides with the application of the typed version. Since functors and natural transformations also act on the level of morphisms one also has to treat this additional information.

Problem 12. How can one check critical pairs for a rewriting system describing categorical notions taking the level of objects into account? A morphism now becomes a type depending on two objects – source and target.

Solution 13. On the level of objects usually a simple test of equality is done to check the compatibility of morphisms. Thus the unification mechanism which is present for rewriting can be exploited to perform this check, too. The typing information has to be encoded with a new function symbol representing the dependent typing. In the case of categories this looks like "$mor(f, a, b)$" where f is the untyped form of the morphism and a, b are source and target, resp.

Example 1. As an example we consider a computational comonad in the presentation which yields the canonical system. As the way to present our example we choose the specification language of the Larch Prover. The sort and variable definitions for the untyped case are:

```
declare sort M          % morphisms
declare variables f, g, h: M
declare operators
        id: -> M        % identity
        *: M,M -> M     % composition
        counit: -> M    % counit
        costar: M -> M  % Kleisli
```

For the typed case the definitions look like this:

```
declare sort M          % typed morphisms
declare sort M'         % untyped morphisms
declare sort O          % objects
declare variables f',g',h': M'
declare variables o1, o2, o3, o4: O
declare operators
        id: -> M'       % identity
        *: M,M -> M'    % composition
        counit: -> M'   % counit
        costar: M -> M' % Kleisli
        mor: M',O,O-> M % explicit typing
        t': O -> O      % functor on object level
```

The following table demonstrates the translation from the untyped case into the typed case.

morphism	untyped version	typed version
$f_{O1 \to O2}$	`f`	`mor(f, o1, o2)`
id_{O1}	`id`	`mor(id, o1, o1)`
$f_{O2 \to O3} \circ g_{O1 \to O2}$	`f * g`	`mor(mor(f,o2,o3)*mor(g,o1,o2),o1,o3)`
v_{O1}	`counit`	`mor(counit, t'(o1), o1)`
$f^{\star}_{T(O1) \to O2}$	`costar(f)`	`mor(costar(mor(f, t'(o1), o2)), t'(o1), t'(o2))`

Exploiting this translation the rewriting system of the computational comonad was verified again respecting the typing. This can be seen in the appendix.

Remark. All the typing information had to be hand-coded which is very error-prone. Unfortunately the Larch Prover did not support this kind of type processing. Therefore we implemented the final decision procedures for comonads and computational comonads in the logic programming language Elf which supports dependent types directly.

5 Implementation in Elf and Application

Here it is shown how the canonical system for computational comonads can be applied where examples are taken from [BG91]. The actual run of the test can be found in the appendix.

Because of the difficulty to represent dependent types in a conventional rewriting tool like the Larch Prover we implemented the rewriting in the logic programming language Elf. We think that this approach has several advantages:

- Types help to encode morphisms correctly but also represent judgements via the "propositions as types principle".
- Elf does not only give an answer substitution but also a term representing the proof which can be used for further inspection.
- Since it is a programming language several rewriting strategies which fit the problem can be implemented.
- Elf allows additionally the treatment of higher order rewriting since the language supports higher order types.

- It is also possible to formulate the concept of critical overlaps in the language.
- Elf allows to prove meta-theorems as the soundness of rewriting with respect to an axiomatic definition of the equality.

An Elf program is split into a static and a dynamic part. The former is only used for type checking whereas the latter is used for proof search. In the appendix both these parts of the program are shown. The dynamic part makes already use of the definitions in the static part so that only in a few cases the type has to be made more explicit.

The sample queries which can be seen in the trace correspond to [BG91]. There a pair of functors is defined relating the category C and the Kleisli category K. We have:

$$alg : C \to K \text{ defined by } alg(f_{A \to B}) := f \circ \epsilon_A$$
$$fun : K \to C \text{ defined by } fun(f_{T(A) \to B}) := f \circ \gamma_A$$

The queries are the test whether alg and fun are indeed functors. Furthermore the following equalities were checked:

$$fun \circ alg = I_C$$
$$alg \circ fun =^e I_K \text{ where } f =^e g \iff fun(f) = fun(g)$$

The test of equality proceeds in two steps where the first one does the translation into the representation which can be used for normalization and the second one does the normalization. The knowledge about the definition of alg and fun is already coded in the translation process.

6 An Incomplete Attempt: Rewriting over a Congruence

In this section we sketch another attempt to achieve a canonical system for comonads which should be extended to computational comonads as well.

Our attempt was motivated by trying to remain within the given specification of a comonad in terms of $(id, \circ, T, \epsilon, \delta)$. Our approach was to take the equational specification of a comonad as presented above, to orient all the equations from left to right, but to work over the congruence generated by the equations 3 for associativity and 5 for the endofunctor.

Working over this congruence results in finite congruence classes which can be computed and have a canonical representation when orienting both rules from left to right. Thus the reducibility of terms is decidable cf. [Bac91]. Furthermore it is easy to find a polynomial interpretation which is decreasing on the rules and remains constant for the congruence. The hard part here is to show confluence.

Neither the left-linear rule method by Huet (cf. [Hue80]) succeeded although the rules are left-linear nor the existence of a unification algorithm for a homomorphism (equation 5) as given in [Vog78] but without associativity did suffice.

Nevertheless the structure of the rules is rather simple since the composition occurs there only once. Besides the usual critical overlaps new ones have to be considered due to the congruence as for example:

$$T(f) \circ \epsilon_T \circ \delta \leftarrow \epsilon_T \circ T(T(f)) \circ \delta \to \epsilon_T \circ \delta \circ T(f)$$

We attempted to formalize this with rules involving exponentiation of the endofunctor as:

$$T^n(\epsilon_B) \circ T^{n+1}(f_{A \to B}) \longrightarrow T^n(f) \circ T^n(\epsilon_A)$$
$$T^{n+2}(f_{A \to B}) \circ T^n(\delta_A) \longrightarrow T^n(\delta_B) \circ T^{n+1}(f)$$

where each such rule comes together with the completed version from associativity. But this really required the treatment of natural numbers and addition which complicates the situation since AC-unification becomes necessary. On the other hand exponents are always ground in a concrete decision problem.

Since the amount of rules was constantly growing we finally looked for a more appropriate formulation of the theory of a comonad although the structure of the rules for a computational comonad follow the same pattern. It is possible that a formalization with the ReDuX [Bün93] system which is able to handle inductive completion also in the presence of AC-operators may succeed. In the literature there several new methods are presented which try to handle to handle infinite sets of rules arising from completion.

7 Conclusions and Future Work

The uniform word problem of computational comonads was shown to be decidable by extending an appropriate canonical system of comonads. The critical pair check had to take the presence of types into account which come from source and target of a morphism. This suggested to use the logic programming language Elf to implement typed rewriting because of its direct support of dependent types. Furthermore the user has direct influence on the strategy for rewriting. For future work we want to continue in two directions – a theoretical and a practical.

The notion of a monad as used in functional programming [Wad93] still has to be investigated. Since this requires the treatment of higher order rewriting [Nip91] Elf is still a suitable tool since it supports higher order types. This would give the right frame to reason about certain monadic functional programs.

In practice it would be very helpful to allow diagrams as a compact visual encoding of equations as input to the prover. The output could also be displayed in an appropriate form. In [FS90] one can already find a suitable graphical language used in the context of categories.

Acknowledgements

I am indebted to Frank Pfenning for his hospitality during a visit to CMU which made it possible to learn more about the programming language Elf and which allowed me fruitful discussions of these ideas. Furthermore I wish to thank Andrzej Filinski for providing me with more insight to monads. Also I am very grateful to my advisor Jochen Pfalzgraf for constant encouragement.

References

[Bac91] L. Bachmair. *Canonical Equational Proofs.* Birkhäuser, 1991.

[BG91] S. Brookes and S. Geva. Computational Comonads and Intentional Semantics. Technical Report CMU-CS-91-190, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1991.

[Bün93] R. Bündgen. Reduce the Redex → ReDuX. In C. Kirchner, editor, *Rewriting Techniques and Applications*, number 690 in Lecture Notes in Computer Science, pages 446–450. Springer-Verlag, 1993.

[BW85] M. Barr and C. Wells. *Toposes, Triples and Theories.* Number 278 in Grundlehren der mathematischen Wissenschaften. Springer-Verlag, 1985.

[FS90] P.J. Freyd and A. Sčedrov. *Categories, Allegories.* Elsevier Science Publishers, 1990.

[Geh94] W. Gehrke. Proof of the Decidability of the Uniform Word Problem for Monads Assisted by Elf. Technical Report 94-66, RISC, 1994.

[GG91] S.J. Garland and J.V. Guttag. *A Guide to LP, The Larch Prover.* MIT, 1991.

[Hue80] G. Huet. Confluent Reductions. *Journal of the Association for Computing Machinery*, 24(4):797–821, October 1980.

[Hue90a] G. Huet. Cartesian closed categories and lambda-calculus. In *[Hue90b]*, pages 7–23. Addison Wesley, 1990.

[Hue90b] G. Huet. *Logical Foundations of Functional Programming.* University of Texas at Austin Programming Series. Addison Wesley, 1990.

[KZ89] D. Kapur and H. Zhang. *RRL: Rewrite Rule Laboratory User's Manual*, revised edition, May 1989.

[Lan79] D.S. Lankford. On proving term rewriting systems are Noetherian. Technical report, Louisiana Technical University, Ruston, LA, 1979.

[Man76] E. Manes. *Algebraic Theories.* Number 26 in Graduate Texts in Mathematics. Springer-Verlag, 1976.

[ML71] S. Mac Lane. *Categories for the Working Mathematician.* Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.

[Mog89] E. Moggi. Computational Lambda-calculus and Monads. In *Fourth Annual Symposium on Logic in Computer Science*, pages 14–23. IEEE, June 1989.

[Nip91] T. Nipkow. Higher Order Critical Pairs. In *Sixth Annual Symposium on Logic in Computer Science*, pages 342–349. IEEE, July 1991.

[Pfe89] F. Pfenning. Elf: a language for verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE, June 1989.

[Vog78] E. Vogel. Unifikation von Morphismen (in German). Diplomarbeit, Universität Karlsruhe, 1978.

[Wad93] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and System Sciences*, pages 233–264. Springer-Verlag, 1993.

Appendix

A LP Traces

Here the check of termination and confluence with the Larch Prover are shown. For termination the typing has not been taken into account, i.e. the termination argument does not make use of types. Afterwards the check of critical pairs is repeated in a setting which also includes types.

A.1 Termination in LP

```
Larch Prover (17 March 1993) logging on 7 October 1994 17:55:44 to
'/usr0/wgehrke/Data/lp/comp_comonad/termination.lplog'.
LP2: execute termination
LP2.1: declare sort M          % morphisms
LP2.2: declare variables f, g, h: M
LP2.3: declare operators
        id: -> M        % identity
        *: M,M -> M     % composition
        counit: -> M    % counit
        costar: M -> M  % Kleisli
..
LP2.4: set ordering polynomial 3
The ordering-method is now 'polynomial 3'.
LP2.5: register polynomial     id      1
LP2.6: register polynomial     *       x + y, 2 * x * y, 2 * x + y
LP2.7: register polynomial     counit  1
LP2.8: register polynomial     costar  x + 2
LP2.9: assert
%   category
        id * f == f
        f * id == f
        (f * g) * h == f * (g * h)
%   comonad
        counit * costar(f) == f
        costar(counit) == id
        costar(f) * costar(g) == costar(f * costar(g))
..
Added 6 equations named user.1, ..., user.6 to the system.
The system now contains 6 rewrite rules.
LP2.10: complete
The following equations are critical pairs between rewrite rules user.4 and
user.3.
  user.7: f * h == counit * (costar(f) * h)
The system now contains 1 equation and 6 rewrite rules.
The following equations are critical pairs between rewrite rules user.6 and
user.3.
  user.8: costar(f * costar(g)) * h == costar(f) * (costar(g) * h)
The system now contains 1 equation and 7 rewrite rules.
The system now contains 8 rewrite rules.
The system is complete.
LP2.11: display
Rewrite rules:
user.1: id * f -> f
user.2: f * id -> f
user.3: (f * g) * h -> f * (g * h)
user.4: counit * costar(f) -> f
user.5: costar(counit) -> id
```

```
user.6: costar(f) * costar(g) -> costar(f * costar(g))
user.7: counit * (costar(f) * h) -> f * h
user.8: costar(f) * (costar(g) * h) -> costar(f * costar(g)) * h
LP2.12: declare operators
        gamma: -> M     % computational comonad
..
LP2.13: register polynomial     gamma   1
LP2.14: assert
        counit * gamma == id
        costar(f) * gamma == gamma * (f * gamma)
..
Added 2 equations named user.9, user.10 to the system.
The system now contains 10 rewrite rules.
LP2.15: complete
The following equations are critical pairs between rewrite rules user.9 and
user.3.
  user.11: h == counit * (gamma * h)
The system now contains 1 equation and 10 rewrite rules.
The following equations are critical pairs between rewrite rules user.10 and
user.3.
  user.12: gamma * (f * (gamma * h)) == costar(f) * (gamma * h)
The system now contains 1 equation and 11 rewrite rules.
The system now contains 12 rewrite rules.
The system is complete.
LP2.16: display
Rewrite rules:
user.1:  id * f -> f
user.2:  f * id -> f
user.3:  (f * g) * h -> f * (g * h)
user.4:  counit * costar(f) -> f
user.5:  costar(counit) -> id
user.6:  costar(f) * costar(g) -> costar(f * costar(g))
user.7:  counit * (costar(f) * h) -> f * h
user.8:  costar(f) * (costar(g) * h) -> costar(f * costar(g)) * h
user.9:  counit * gamma -> id
user.10: costar(f) * gamma -> gamma * (f * gamma)
user.11: counit * (gamma * h) -> h
user.12: costar(f) * (gamma * h) -> gamma * (f * (gamma * h))
End of input from file '/usr0/wgehrke/Data/lp/comp_comonad/termination.lp'.
LP3: quit
```

A.2 Typing in LP

```
Larch Prover (17 March 1993) logging on 7 October 1994 17:56:02 to
'/usr0/wgehrke/Data/lp/comp_comonad/typing.lplog'.
LP2: execute typing
LP2.1: declare sort M          % typed morphisms
LP2.2: declare sort M'         % untyped morphisms
LP2.3: declare sort O          % objects
LP2.4: declare variables f',g',h': M'
LP2.5: declare variables o1, o2, o3, o4: O
LP2.6: declare operators
        id: -> M'       % identity
        *: M,M -> M'    % composition
        counit: -> M'   % counit
        costar: M -> M' % Kleisli
        mor: M',O,O-> M % explicit typing
        t': O -> O      % action of the functor on objects
```

```
        gamma: -> M'    % computational comonad
..
LP2.7: set ordering left-to-right
The ordering-method is now 'left-to-right'.
LP2.8: assert
%   category
        mor(mor(id, o2, o2) * mor(f', o1, o2), o1, o2) == mor(f', o1, o2)
        mor(mor(f', o1, o2) * mor(id, o1, o1), o1, o2) == mor(f', o1, o2)
        mor(mor(mor(f', o3, o4) * mor(g', o2, o3), o2, o4)
          * mor(h', o1, o2), o1, o4)
        == mor(mor(f', o3, o4) *
          mor(mor(g', o2, o3) * mor(h', o1, o2), o1, o3), o1, o4)
%   comonad
        mor(mor(counit, t'(o2), o2)
          * mor(costar(mor(f', t'(o1), o2)), t'(o1), t'(o2)), t'(o1), o2)
        == mor(f', t'(o1), o2)
        mor(costar(mor(counit, t'(o1), o1)), t'(o1), t'(o1))
        == mor(id, t'(o1), t'(o1))
        mor(mor(costar(mor(f', t'(o2), o3)), t'(o2), t'(o3))
          * mor(costar(mor(g', t'(o1), o2)), t'(o1), t'(o2)), t'(o1), t'(o3))
        == mor(costar(mor(mor(f', t'(o2), o3)
          * mor(costar(mor(g', t'(o1), o2)), t'(o1), t'(o2)),
            t'(o1), o3)), t'(o1), t'(o3))
%   completed rules
        mor(mor(counit, t'(o3), o3)
          * mor(mor(costar(mor(f', t'(o2), o3)), t'(o2), t'(o3))
            * mor(g', o1, t'(o2)), o1, t'(o3)), o1, o3)
        == mor(mor(f', t'(o2), o3)
            * mor(g', o1, t'(o2)), o1, o3)
        mor(mor(costar(mor(f', t'(o3), o4)), t'(o3), t'(o4))
            * mor(mor(costar(mor(g', t'(o2), o3)), t'(o2), t'(o3))
              * mor(h', o1, t'(o2)), o1, t'(o3)), o1, t'(o4))
        == mor(mor(costar(mor(mor(f', t'(o3), o4)
            * mor(costar(mor(g', t'(o2), o3)), t'(o2), t'(o3)), t'(o2), o4))
              , t'(o2), t'(o4))
                * mor(h', o1, t'(o2)), o1, t'(o4))
..
Added 8 equations named user.1, ..., user.8 to the system.
The system now contains 8 rewrite rules.
LP2.9: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.10: assert
%   computational comonad
        mor(mor(counit, t'(o1), o1) * mor(gamma, o1, t'(o1)), o1, o1)
        == mor(id, o1, o1)
        mor(mor(costar(mor(f', t'(o1), o2)), t'(o1), t'(o2))
          * mor(gamma, o1, t'(o1)), o1, t'(o2))
        == mor(mor(gamma, o2, t'(o2)) *
          mor(mor(f', t'(o1), o2) * mor(gamma, o1, t'(o1)), o1, o2),
            o1, t'(o2))
%   completed rules
        mor(mor(counit, t'(o2), o2) *
          mor(mor(gamma, o2, t'(o2)) * mor(g', o1, o2), o1, t'(o2)),
            o1, o2)
        == mor(g', o1, o2)
        mor(mor(costar(mor(f', t'(o2), o3)), t'(o2), t'(o3))
          * mor(mor(gamma, o2, t'(o2)) * mor(g', o1, o2), o1, t'(o2)),
```

```
          o1, t'(o3))
       == mor(mor(gamma, o3, t'(o3)) *
          mor(mor(f', t'(o2), o3) *
            mor(mor(gamma, o2, t'(o2)) * mor(g', o1, o2),
              o1, t'(o2)), o1, o3), o1, t'(o3))
..
Added 4 equations named user.9, ..., user.12 to the system.
The system now contains 12 rewrite rules.
LP2.11: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
End of input from file '/usr0/wgehrke/Data/lp/comp_comonad/typing.lp'.
LP3: quit
```

B Elf traces

All parts of the Elf program are presented to illustrate the usage of Elf for rewriting making use of dependent types. The trace shows the automated proof of examples taken from [BG91]. The current implementation of Elf is embedded into an image of the SML/NJ compiler and it is accessible through functions from the top level.

B.1 The Static Part of the Elf Program

```
%%% category
obj    : type. %name obj O
mor    : obj -> obj -> type. %name mor M
id     : {A:obj} mor A A.
*      : mor O2 O3 -> mor O1 O2 -> mor O1 O3. %infix right 10 *
%%% comonad description with (T,eps,del)
T'     : obj -> obj.
T      : mor A B -> mor (T' A) (T' B).
eps    : {A:obj} mor (T' A) A.
del    : {A:obj} mor (T' A) (T' (T' A)).
%%% comonad description with counit and costar via Kleisli category
counit: {A:obj} mor (T' A) A.
costar: mor (T' A) B -> mor (T' A) (T' B).
%%% missing ingredient for a computational comonad
gamma : {A:obj} mor A (T' A).
%%% for checks from paper by Brookes and Geva
alg    : mor A B -> mor (T' A) B.
fun    : mor (T' A) B -> mor A B.
```

B.2 The Dynamic Part of the Elf Program

```
%%% rewriting for computational comonads in the Kleisli category presentation
rule   : mor A B -> mor A B -> type.
comon1: rule ((id B) * F) F.
comon2: rule (F * (id A)) F.
comon3: rule ((F * G) * H) (F * (G * H)).
comon4: rule ((counit B) * (costar F)) F.
comon5: rule (costar (counit A)) (id (T' A)).
comon6: rule ((costar F) * (costar G)) (costar (F * (costar G))).
comon7: rule ((counit B) * ((costar F) * G)) (F * G).
comon8: rule ((costar F) * ((costar G) * H)) ((costar (F * (costar G))) * H).
comp1 : rule ((counit A) * (gamma A)) (id A).
comp2 : rule ((costar F) * (gamma A)) ((gamma B) * (F * (gamma A))).
```

```
comp3 : rule ((counit B) * ((gamma B) * F)) F.
comp4 : rule ((costar F) * ((gamma A) * G))
        ((gamma B) * (F * ((gamma A) * G))).
%%% rewrite relation for computational comonad
rew   : mor A B -> mor A B -> type.   % try to rewrite
step  : mor A B -> mor A B -> type.   % do at least one rewrite
simple: step F F''
        <- rule F F' <- rew F' F''.
step*1: step (F * G) H
        <- step F F' <- rew (F' * G) H.
step*2: step (F * G) H
        <- step G G' <- rew (F * G') H.
step^*: step (costar F) H
        <- step F F' <- rew (costar F') H.
try   : rew F F''
        <- step F F' <- rew F' F''.
fini  : rew F F.
%%  dynamic equality of morphisms over (id, *, counit, costar, gamma)
==    : mor A B -> mor A B -> type. %name == EQ
%infix none 8 ==
moreq : F == G
        <- rew F H <- rew G H.
%%% decision procedure for (computational) comonads
%%  translation of morphisms with the help of the Kleisli category
trans : mor A B -> mor A B -> type.
tr_*  : trans (F * G) (F' * G')
        <- trans F F' <- trans G G'.
tr_^* : trans (costar F) (costar F')
        <- trans F F'.
tr_alg: trans (alg (F : mor A B)) (F' * (counit A))
        <- trans F F'.
tr_fun: trans (fun (F : mor (T' A) B)) (F' * (gamma A))
        <- trans F F'.
tr_T  : trans (T (F : mor A B)) (costar (F' * (counit A)))
        <- trans F F'.
tr_e  : trans (eps A) (counit A).
tr_d  : trans (del A) (costar (id (T' A))).
tr_0  : trans F F.
%%  dynamic equality of morphisms over (id,*,counit,costar,gamma,T,eps,del,alg,fun)
===   : mor A B -> mor A B -> type. %name === EQU
%infix none 8 ===
cmoneq: F === G
        <- trans F F' <- trans G G' <- F' == G'.
```

B.3 A Sample Run

```
Standard ML of New Jersey, Version 0.93, February 15, 1993
Elf, Version 0.4, July 1, 1993, saved on Mon Oct  3 10:03:50 EDT 1994
val it = () : unit
- initload ["static.elf"]["dynamic.elf"];
  . . .
static.elf --- 1 --- static
dynamic.elf --- 2 --- dynamic
val it = () : unit
- top();
Using: dynamic.elf
Solving for: rule rew step == trans ===
?- {A} (alg (id A)) === (counit A).                    % alg on id
```

```
Solving...
solved
yes
?- {O1}{O2}{O3}{F : mor O2 O3}{G : mor O1 O2}
(alg (F * G)) === (alg F) * (costar (alg G)).          % alg on *
   Solving...
solved
yes
?- {A} (fun (counit A)) === (id A).                    % fun on id'
Solving...
solved
yes
?- {O1}{O2}{O3}{F : mor (T' O2) O3}{G : mor (T' O1) O2}
(fun (F * (costar G))) === (fun F) * (fun G).          % fun on *'
   Solving...
solved
yes
?- {A}{B}{F : mor A B} (fun (alg F)) === F.
Solving...
solved
yes
?- {A}{B}{F : mor (T' A) B} (fun (alg (fun F))) === (fun F).
Solving...
solved
yes
?- {A} ((T (eps A)) * ((del A) * (gamma A))) === (gamma A). % another positive test
Solving...
solved
yes
?- {A:obj} (T (gamma A)) === (del A).                  % test if refused
Solving...
no
?- {F}{G} (F * G) == (G * F).                          % another test if refused
Solving...
no
?-
```

This article was processed using the LaTeX macro package with LLNCS style

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890